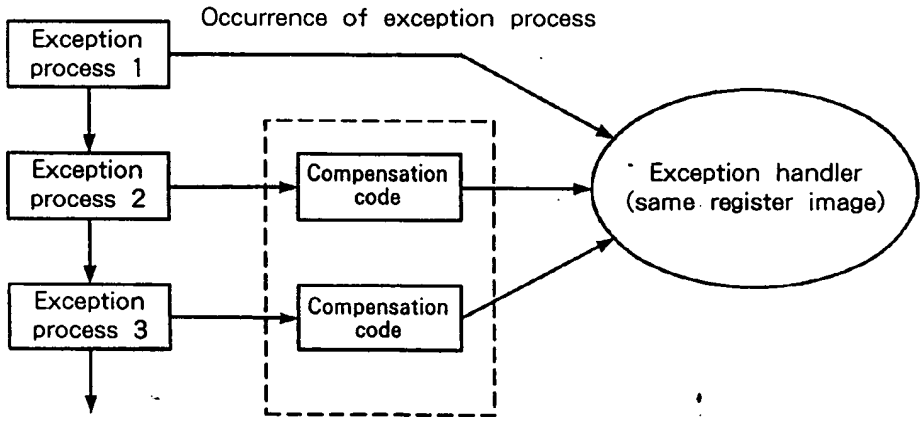
Occurrence of exception process
Exception process 1
Exception process 2
Exception process 3
Compensation code
Compensation code
Exception handler (same register image)

Fig. 1

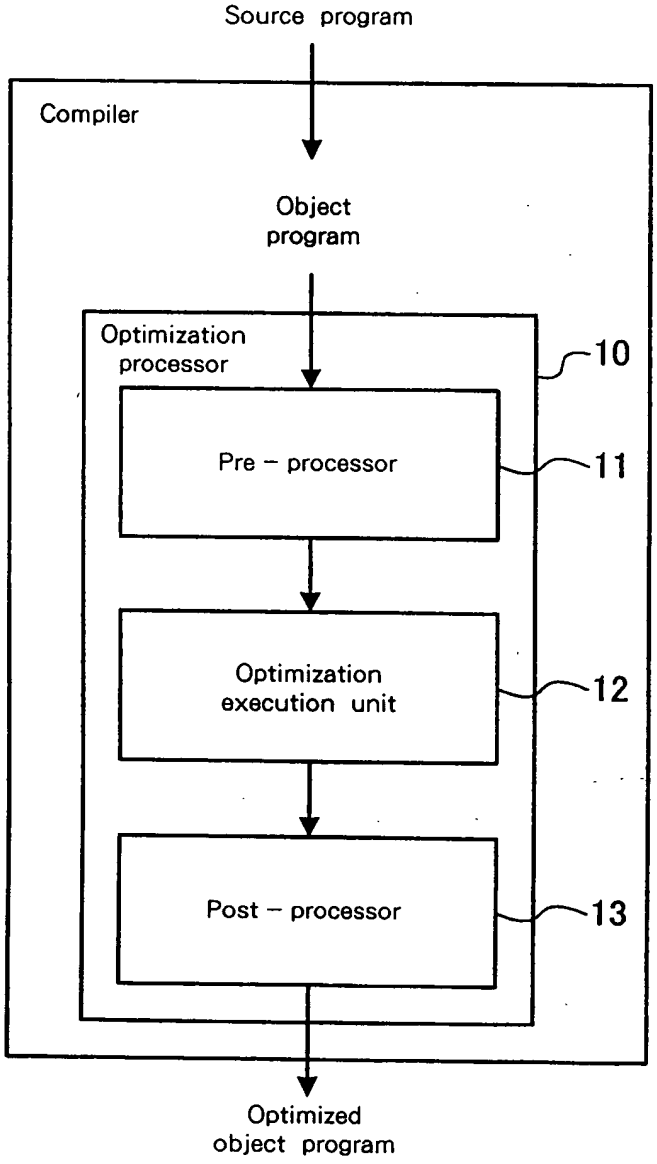
Source program
Compiler
Object program
Optimization processor
10
Pre – processor
11
Optimization execution unit
12
Post – processor
13
Optimized object program

Fig. 2

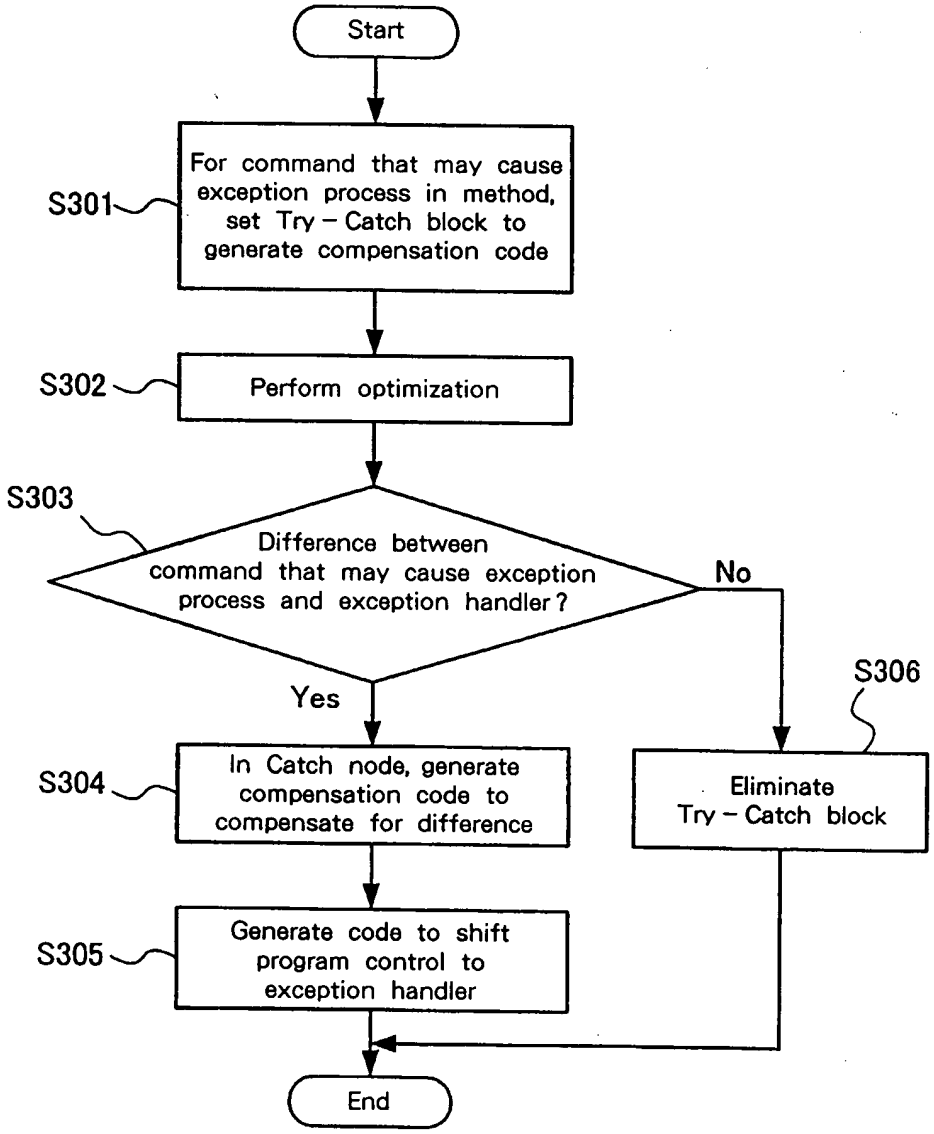
Start
For command that may cause exception process in method, set Try – Catch block to generate compensation code
S301
Perform optimization
S302
S303
Difference between command that may cause exception process and exception handler?
No
Yes
S306
In Catch node, generate compensation code to compensate for difference
S304
Eliminate Try – Catch block
Generate code to shift program control to exception handler
S305
End

Fig. 3

```
for (repeat all basic blocks) {
    bbidx = index of basic block
    if (this basic block is the originating point for an exception handler) {
        reach[bbidx].gen = (all TRYs and all local variables corresponding
        to the exception handler);
    } else {
        reach[bbidx].gen = φ
    }
    reach[bbidx].kill = φ
    for (repeat all commands in the basic block in the execution order) {
        if (command == write to local variable) {
            reach[bbidx].kill U= (all TRYs and written local variable numbers);
        }
    }
}
```

Fig. 4

$$reach[B].in = \left(\bigcup_{P \in Pred(B)} reach[P].out \right) \cup reach[B].gen$$

$$reach[B].out = reach[B].in - reach[B].kill$$

Fig. 5

```
for  (repeat all TRYs) USE_VIA_EH_LIST[TRY] = φ;
for  (repeat all basic blocks) {
   bbidx = index of basic block ;
   if (reach[bbidx]. in != φ) {
      for  (repeat all commands in the basic block in the execution order) {
         switch  (command) {
         case write to a local variable :
            reach[bbidx]. in -= (all TRYs and written local variable numbers) ;
            break;
         case read from a local variable :
            for  (repeat all TRYs) {
               if  ((TRY, a read local variable number) ∈ reach[bbidx]. in)  {
                  USE_VIA_EH_LIST[TRY]  U= read local variable number ;
               }
            }
            break;
         }
      }
   }
}
```

Fig. 6

```
for  (repeat all basic blocks) {
   if  (basic block is included in a try region) {
      TRY = identification number for a Try Region
      if (USE_VIA_EH_LIST[TRY] != φ) {
         bbidx = index of basic block ;
         for  (repeat all commands in the basic block in the execution order) {
            if  (command ∈ command that may cause exception) {
               if  (exception handler is present that corresponds to exception that may occur) {
                  < set new exception handler for this command >
               }
            }
         }
      }
   }
}
```

Fig. 7

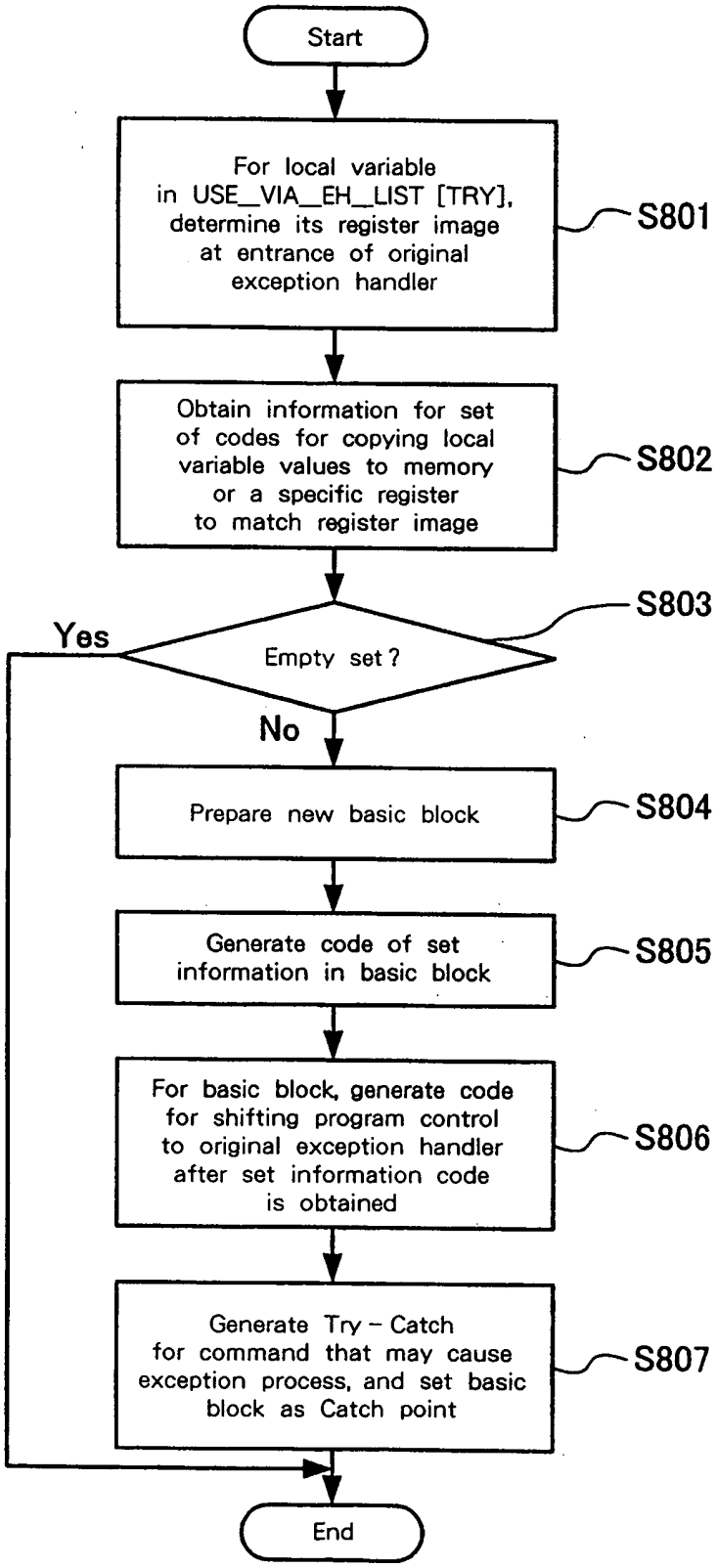
Start
For local variable in USE_VIA_EH_LIST [TRY], determine its register image at entrance of original exception handler
S801
Obtain information for set of codes for copying local variable values to memory or a specific register to match register image
S802
Empty set ?
S803
Yes
No
Prepare new basic block
S804
Generate code of set information in basic block
S805
For basic block, generate code for shifting program control to original exception handler after set information code is obtained
S806
Generate Try – Catch for command that may cause exception process, and set basic block as Catch point
S807
End

Fig. 8

```
int MIN_VAL, MAX_VAL;
void sample(int a[][], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          --- (1)
    j = 0;          --- (2)
    try {
        min = a[i][j];    --- (3)
        max = min;        --- (4)
        i = 0;            --- (5)
        while(i < size_x) {         --- (6)
            j = 0;                  --- (7)
            while(j < size_y) {     --- (8)
                int val = a[i][j];          --- (9)
                if (min > val) {    --- (10)
                    min = val;      --- (11)
                }
                if (max < val) {    --- (12)
                    max = val;      --- (13)
                }
                j++;                --- (14)
            }
            i++;            --- (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("ArrayIndexOutOfBounds i=" + i + " j=" + j);     --- (16)
        /* error status : min > max */
        max = 0x80000000;--- (17)
        min = 0x7FFFFFFF;--- (18)
    }
    MIN_VAL = min;                  --- (19)
    MAX_VAL = max;                  --- (20)
}
```

Fig. 9

```
int MIN_VAL, MAX_VAL;
void sample(int a[][], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          — (1)
    j = 0;          — (2)
    try {
        NULLCHECK a;                    — (3.1)
        try {
            SIZECHECK i, a[];           — (3.2)
        } catch(ArrayIndexOutOfBoundsException e) {
            copy i and j variable values to R1 and R2
            goto Handler;
        }
        try {
            SIZECHECK j, a[i];          — (3.3)
        } catch(ArrayIndexOutOfBoundsException e) {
            copy i and j variable values to R1 and R2
            goto Handler;
        }
        min = a[i][j];          — (3.4)
        max = min; — (4)
        i = 0;                  — (5)
        while(i < size_x) {     — (6)
            j = 0;              — (7)
            while(j < size_y) {         — (8)
                try {
                    SIZECHECK i, a[]; — (9.1)
                } catch(ArrayIndexOutOfBoundsException e) {
                    copy i and j variable values to R1 and R2
                    goto Handler;
                }
                try {
                    SIZECHECK j, a[i];          — (9.2)
                } catch(ArrayIndexOutOfBoundsException e) {
                    copy i and j variable values to R1 and R2
                    goto Handler;
                }
                int val = a[i][j];      — (9.3)
                if (min > val) {        — (10)
                    min = val;          — (11)
                }
                if (max < val) {        — (12)
                    max = val;          — (13)
                }
                j++;                    — (14)
            }
            i++;                — (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
    Handler:
        /* i, j variable values are stored in R1 and R2 */
        System.err.println("ArrayIndexOutOfBounds i=" + i + " j=" + j);    — (16)
        /* error status : min > max */
        max = 0x80000000;       — (17)
        min = 0x7FFFFFFF;       — (18)
    }
    MIN_VAL = min;              — (19)
    MAX_VAL = max;              — (20)
}
```

Fig. 10

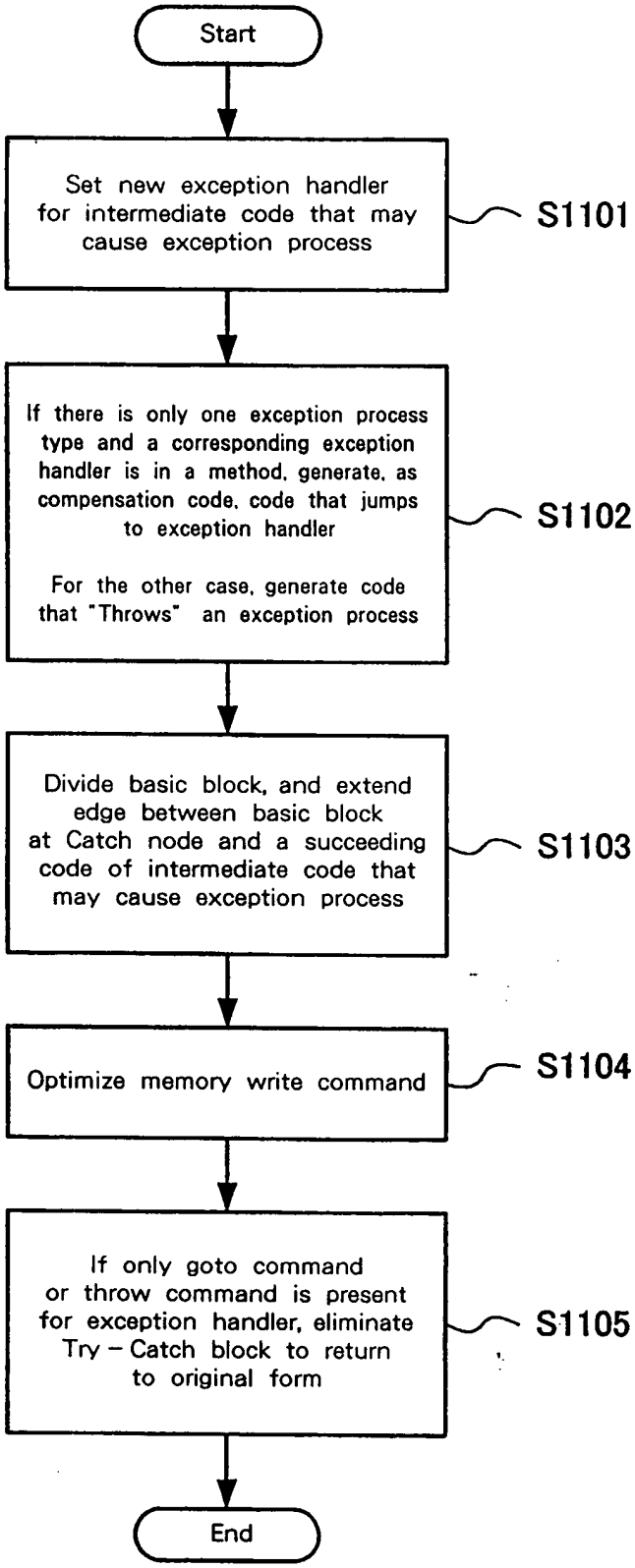
Start
Set new exception handler for intermediate code that may cause exception process
S1101
If there is only one exception process type and a corresponding exception handler is in a method, generate, as compensation code, code that jumps to exception handler
For the other case, generate code that "Throws" an exception process
S1102
Divide basic block, and extend edge between basic block at Catch node and a succeeding code of intermediate code that may cause exception process
S1103
Optimize memory write command
S1104
If only goto command or throw command is present for exception handler, eliminate Try – Catch block to return to original form
S1105
End

Fig. 11

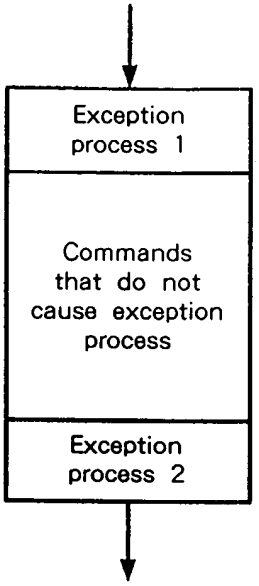
Exception process 1
Commands that do not cause exception process
Exception process 2

Fig. 12

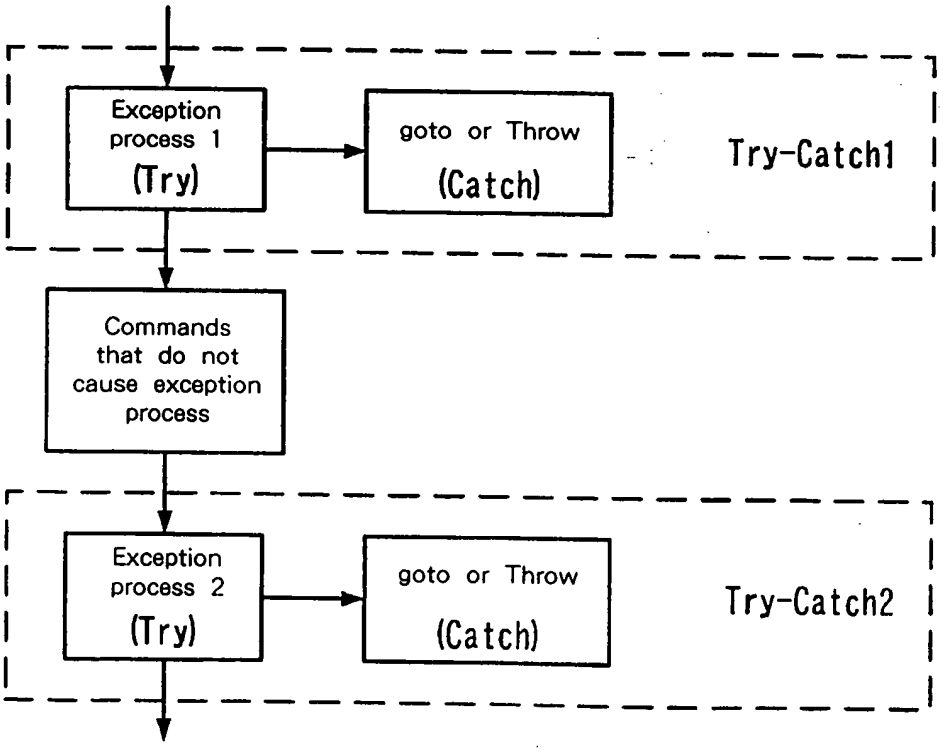
Exception process 1 (Try)
goto or Throw (Catch)
Try-Catch1
Commands that do not cause exception process
Exception process 2 (Try)
goto or Throw (Catch)
Try-Catch2

Fig. 13

```
static int mem_pos;       // variable in a memory
static int mem_a[ ];      // variable in a memory
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while(mem_a[mem_pos] != reg_data) {
        mem_pos ++;
    }
}
```

Fig. 14

```
static int mem_pos;       // variable in a memory
static int mem_a[ ];      // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;          // variable in a register
    int reg_a[ ];         // variable in a register

    mem_pos = 0;

    goto entry;
loop:
    reg_pos = mem_pos;
    reg_pos ++;
    mem_pos = reg_pos;
entry:
    reg_pos = mem_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;              // intermediate code that may cause an exception process
    SIZECHECK reg_pos, reg_a[];   // intermediate code that may cause an exception process
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

Fig. 15

```
static int mem_pos;       // variable in a memory
static int mem_a[ ];      // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;          // variable in a register
    int reg_a[ ];         // variable in a register

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;              // intermediate code that may cause an exception process

    goto entry;                                    1601
loop:
    reg_pos ++;
    mem_pos = reg_pos; -① // this intermediate code can not be moved outside a loop
entry:
    SIZECHECK reg_pos, reg_a[];   // intermediate code that may cause an exception process
    if (reg_a[reg_pos] != reg_data) goto loop;
}
                                                   1602
```

Fig. 16

```
static int mem_pos;       // variable in a memory
static int mem_a[ ];      // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;          // variable in a register
    int reg_a[ ];         // variable in a register

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    try {
        NULLCHECK reg_a;  // intermediate code that may cause an exception process
    } catch(Throwable t) {                                                    } 1701
            throw t;
    }

    goto entry;
loop:
    reg_pos ++;
    mem_pos = reg_pos;
entry:
    try {
        SIZECHECK reg_pos, reg_a[];       // intermediate code that may cause an exception process
                                          //
    } catch(Throwable t) {                                                    } 1702
            throw t;
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

Fig. 17

```
static int mem_pos;       // variable in a memory
static int mem_a[ ];      // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;          // variable in a register
    int reg_a[ ];         // variable in a register

    reg_pos = 0;
    reg_a = mem_a;
    try {
        NULLCHECK reg_a; // intermediate code that may cause an exception process
    } catch(Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }

    goto entry;                    1801
loop:
    reg_pos ++;                                  1802
entry:
    try {
        SIZECHECK reg_pos, reg_a[];      // intermediate code that may cause an exception process
                                         //
    } catch(Throwable t) {
        mem_pos = reg_pos;          -①
        throw t;
    }
    if (reg_a[reg_pos] != reg_data) goto loop;      1803
    mem_pos = reg_pos;    -②
}
```

Fig. 18

```
try {
    ExceptionCheck;       // intermediate code that may cause an exception process
} catch(Throwable t) {
    throw t;
}
```

```
ExceptionCheck;  // intermediate code that may cause an exception process
```

Fig. 19

```
static int mem_pos;       // variable in a memory
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while(func(mem_pos) != reg_data) {
        mem_pos ++;
    }
}
```

Fig. 20

```
static int mem_pos;       // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos; // variable in a register
    int reg_ret; // variable in a register

    reg_pos = 0;

    goto entry;
loop:
    reg_pos ++;          <-- 2101
entry:
    try {
        reg_ret = func(reg_pos);          <-- 2102
    } catch(Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }
    if (reg_ret != reg_data) goto loop;          <-- 2103
    mem_pos = reg_pos;
}
```

Fig. 21

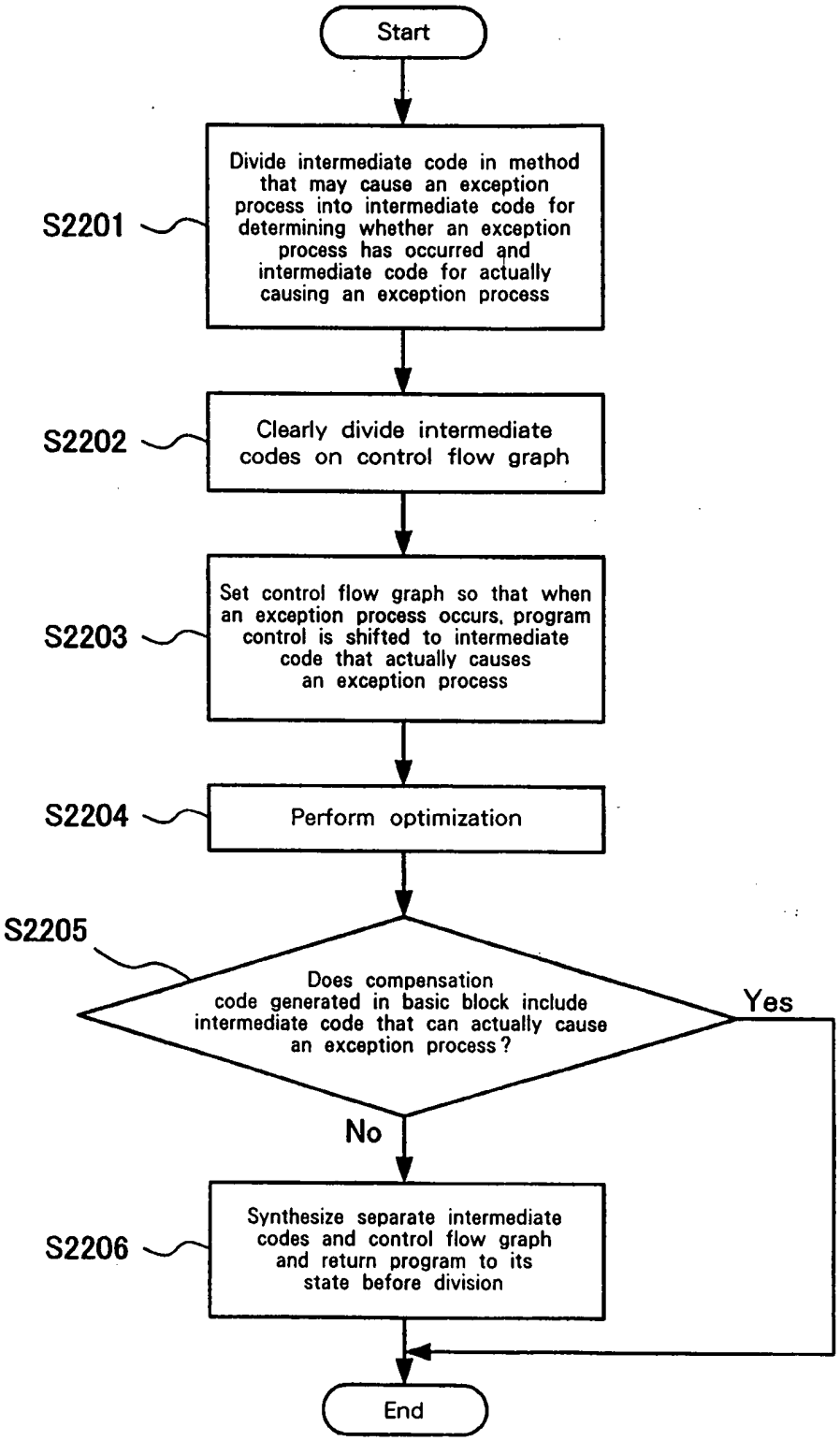
Start
Divide intermediate code in method that may cause an exception process into intermediate code for determining whether an exception process has occurred and intermediate code for actually causing an exception process
S2201
Clearly divide intermediate codes on control flow graph
S2202
Set control flow graph so that when an exception process occurs, program control is shifted to intermediate code that actually causes an exception process
S2203
Perform optimization
S2204
S2205
Does compensation code generated in basic block include intermediate code that can actually cause an exception process?
Yes
No
Synthesize separate intermediate codes and control flow graph and return program to its state before division
S2206
End

Fig. 22

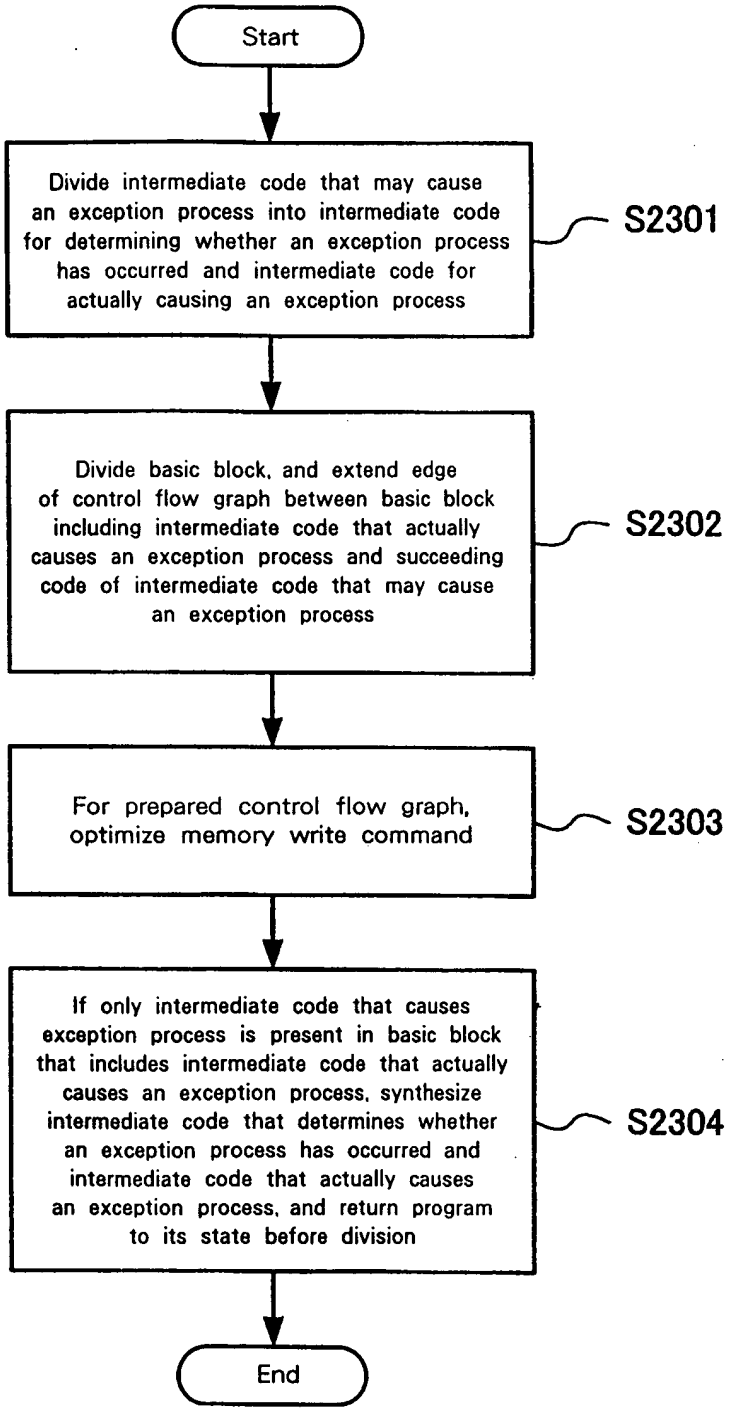
Start
Divide intermediate code that may cause an exception process into intermediate code for determining whether an exception process has occurred and intermediate code for actually causing an exception process
S2301
Divide basic block, and extend edge of control flow graph between basic block including intermediate code that actually causes an exception process and succeeding code of intermediate code that may cause an exception process
S2302
For prepared control flow graph, optimize memory write command
S2303
If only intermediate code that causes exception process is present in basic block that includes intermediate code that actually causes an exception process, synthesize intermediate code that determines whether an exception process has occurred and intermediate code that actually causes an exception process, and return program to its state before division
S2304
End

Fig. 23

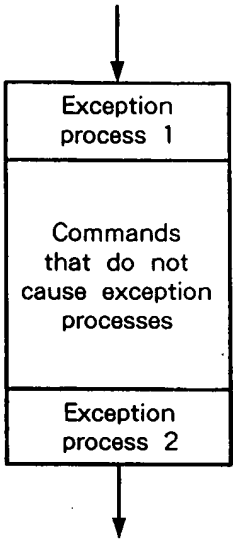
Exception process 1
Commands that do not cause exception processes
Exception process 2

Fig. 24

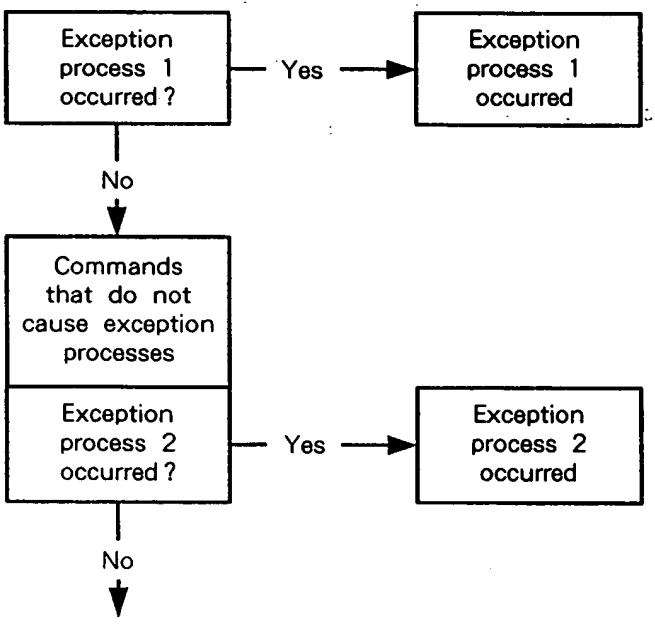
Exception process 1 occurred ?
Yes
Exception process 1 occurred
No
Commands that do not cause exception processes
Exception process 2 occurred ?
Yes
Exception process 2 occurred
No

Fig. 25

```
static int mem_pos;       // variable in a memory
static int mem_a[ ];      // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;          // variable in a register
    int reg_a[ ];         // variable in a register

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) {    // intermediate code that determines
                                   //    whether an exception process will occur
        NULLCHECK_EXCEPTION reg_a;       // intermediate code that causes an exception process
    }

    goto entry;
loop:
    reg_pos ++;                                2601
    mem_pos = reg_pos;
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) {          // intermediate code that determines
                                                    //     whether an exception process will occur
        SIZECHECK_EXCEPTION reg_pos, reg_a[];       // intermediate code that causes an exception process
    }
    if (reg_a[reg_pos] != reg_data) goto loop;      2602
}
```

Fig. 26

```
static int mem_pos;       // variable in a memory
static int mem_a[ ];      // variable in a memory
void SearchPos(int reg_data)
{
    int reg_pos;          // variable in a register
    int reg_a[ ];         // variable in a register

    reg_pos = 0;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) {    // intermediate code that determines
                                   //    whether an exception process will occur
        mem_pos = reg_pos;
        NULLCHECK_EXCEPTION reg_a;        // intermediate code that causes an exception process
    }

    goto entry;
loop:                                              2702
    reg_pos ++;          <-- 2701
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) {         // intermediate code that determines
                                                   //    whether an exception process will occur
        mem_pos = reg_pos;          -①
        SIZECHECK_EXCEPTION reg_pos, reg_a[];      // intermediate code that causes an exception process
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
    mem_pos = reg_pos;    -②
}                                                  2703
```

Fig. 27

```
if (EXCEPTIONCHECK_FAIL) {          // intermediate code that determines
                                        // whether an exception process will occur
    EXCEPTION;                      // intermediate code that causes an exception process
}
```

```
ExceptionCheck; // intermediate code that may cause an exception process
```

Fig. 28